Tutorials for queueing simulations

N.D. Van Foreest and E.R. Van Beesten

January 13, 2020

CONTENTS

1 Tutorial 3: Simulation of the $G/G/1$ queue in continuous time 2

1 TUTORIAL 3: SIMULATION OF THE $G/G/1$ QUEUE IN CONTINUOUS TIME

In this tutorial we will write a simulator for the $G/G/1$ queue. For this we use two concepts that are essential to simulate any stochastic system of reasonable complexity: an *event stack* (or event schedule) to keep track of the sequence in which events occur, and *classes* to organize data and behavior into single logical units. We will work in steps towards our goal; along the way you will learn a number of fundamental and highly interesting concepts such as *classes* and efficient *data structures*. Once you have understood the implementation of the $G/G/1$ queue that we are going to build in this tutorial, discrete event simulators are no longer a black box for you. Hence, what you will learn from this tutorial extends well beyond the simulation of the $G/G/1$ queueing process.

Once we have built our simulator we use it to analyze the queueing behavior at a check-in desk at an airport.

In Section **??** we will generalize the simulator to the $G/G/c$ queue and clean up the code by working with a class for the $G/G/c$ queue. We can then also extend the case accordingly.

1.1 *Sorting with event stacks*

If we simulate a stochastic system we like to move from event to event. To illustrate this idea, consider a queueing process such as the $M/M/1$ queue. It is clear that only at arrival and departure epochs something interesting happens, so that we can neglect any time in between arrival and departure epochs as nothing happens. Hence, in our simulator we prefer not to keep track of the entire time line, but just of the relevant epochs. Once we have mechanism to store and organize these events, we can jump from event to event.

Clearly, we want to follow the sequence of events in the correct order of time. For this we use an *event stack*. To see how this works, we will consider a simple example first, and then we extend to more difficult situations.

Suppose we have 4 students, and we like to sort them in increasing order of age. One way to do this is to insert them into a list such that the insertion process respects the correct ordering right away. For this very general problem (insert elements in an ordered way in a list) a number of efficient data structures has been developed, and one of these is the *heap queue*.

1.1. Search the web on `python` and `heap queue`. Study the examples and write some code to sort the students Jan, Pete, Clair, and Cynthia, with ages 21, 20, 18, and 25, in order of increasing age.

s.1.1. If you have read (and thought about) the documentation on the web, you must have ended up with this code.

```
from heapq import heappop, heappush
import scipy
from scipy.stats import uniform, expon

scipy.random.seed(3)

def sort_ages():
    stack = []

    heappush(stack, (21, "Jan"))
    heappush(stack, (20, "Piet"))
    heappush(stack, (18, "Klara"))
    heappush(stack, (25, "Cynthia"))

    while stack:
        age, name = heappop(stack)
        print(name, age)
```

```
sort_ages()
```

heappush pushes things on the stack in a sorted fashion, heappop takes things from the stack. First we put the students on the stack. To print the student names in sorted order of agee, we remove items from the stack until it is empty.

1.2. Extend the code of the previous exercise such that we can include more information with the ages than just the name, for instance, also the brand of their mobile phone.

s.1.2. We can make the tuples, i.e., the data between the brackets, just longer.

```
def sort_ages_with_more_info():
    stack = []

    heappush(stack, (21, "Jan", "Huawei"))
    heappush(stack, (20, "Piet", "Apple"))
    heappush(stack, (18, "Klara", "Motorola"))
    heappush(stack, (25, "Cynthia", "Nexus"))

    while stack:
        age, name, phone = heappop(stack)
        print(age, name, phone)

sort_ages_with_more_info()
```

Observe that we label the students by their age, and organize the data in the form of tuples.

1.3. It may seem that we have just solved a simple toy problem, but this is not true. In fact, we have established something of real importance: heap queues form the core functionality of (nearly) all discrete event simulators used around the world. To help you understand this fact, develop some ideas how you could to use heap queues to simulate a queueing process.

s.1.3. For a queueing process we can start with putting a number of job arrivals on the stack and label these events as 'arrivals'. Whenever a service starts, compute the departure time of a job, and put this departure moment on the stack. Label this event as a 'departure'. Then move to the next event. Since the event stack is sorted in time, the event at the head of the stack is the first moment in time something useful happens.

More generally, a discrete-time stochastic process moves from event to event. At an event certain actions have to be taken, and these actions may involve the generation of new events or the removal of events that are in the stack. The new events are put on the stack, the obsolete ones removed from the stack, and then the simulator moves to the first event on the stack, actions are taken, move to the next event, and so on.

```
# void  code for numbering
```

1.2 *G/G/1 queue*

In this section we will set up a simulator for the $G/G/1$ queue with an event stack. Let us work in steps toward the final simulator.

We start with some basic imports, initialize the stack, and define two numbers ARRIVAL and DEPARTURE that will be used to tag the type of event that will be popped from the event stack. So, make a new file, and type the following at the top of it.

```
ARRIVAL = 0
DEPARTURE = 1

stack = [] # this is the event stack
```

1.4. Why do we introduce the variables `ARRIVAL` and `DEPARTURE`; we could also use the numbers 0 and 1, and this works just as well?

s.1.4. We use variables for clarity and ease of reading the code. We will simply make less mistakes like this.

First of all we need jobs with arrival times and service times. The easiest way to handle jobs in a simulator is by means of a class, as in the code below.

```
class Job:
    def __init__(self):
        self.arrival_time = 0
        self.service_time = 0
        self.departure_time = 0
        self.queue_length_at_arrival = 0

    def sojourn_time(self):
        return self.departure_time - self.arrival_time

    def waiting_time(self):
        return self.sojourn_time() - self.service_time

    def __repr__(self):
        return f"{self.arrival_time}, {self.service_time}, {self.departure_time}\n"

    def __le__(self, other):
        # this is necessary to sort jobs when they have the same arrival times.
        return self.id <= other.id
```

A class has a number of *attributes*, such as `self.arrival_time`, to capture its state and a number of functions, such as `def waiting_time(self)`, to compute specific information that relates to the class[1]. We initialize the arrival time and service time to zero, and we we use the attributes `departure_time` and `queue_length_at_arrival` for a statistical analysis at the end of the simulation. Finally, we have functions to compute the waiting time and the sojourn time, and `__repr__` to print info about the job. (We also need the function `__le__` for a technical reason, which you do not need to know if you are a python newbie. When two jobs have the same arrival time, the heap queue needs extra information to sort such jobs. For this we simply use `id` of the object.)

Note that our naming of functions also acts as documentation of what the functions do. Note also that member variables and functions in python start with the word `self`; this is to distinguish the (value of the) member variables from variables with the same name but lying outside the scope of the class.

Classes are extremely useful programming concepts, as it enables you to organize state (that is, attributes) and behavior (that is, functions that apply to the attributes) into logical components. Moreover, classes can offer functionality to a programmer without the programmer needing to understand how this functionality is built. Classes offer many more advantages such as inheritance, but we will not discuss that here.

Once we have a class, we can make an object with the code `job = Job()`.

1.5. Check the internet to find out the difference between a 'class' and an 'object'.

s.1.5. The wording is important here. Objects are instances of classes. As an example: Albert Einstein was a human being. Here, 'Einstein' is the object, and 'human being' is a class.

1.6. Make 10 jobs with exponentially distributed inter-arrival times, with $\lambda = 2$, and exponentially distributed service times with $\mu = 3$. Put these jobs on an event stack, and print them in order of arrival time. Tag the events with the job and event type (which is an arrival).

[1] In python, functions belonging to a class are called 'methods' or 'member functions'.

s.1.6. One way is like this.

```
def experiment_1():
    labda = 2.0
    mu = 3.0
    rho = labda / mu
    F = expon(scale=1.0 / labda)  # interarrival time distributon
    G = expon(scale=1.0 / mu)  # service time distributon

    num_jobs = 10

    time = 0
    for i in range(num_jobs):
        time += F.rvs()
        job = Job()
        job.arrival_time = time
        job.service_time = G.rvs()
        heappush(stack, (job.arrival_time, job, ARRIVAL))

    while stack:
        time, job, typ = heappop(stack)
        print(job)



experiment_1()
```

Now that we know how to make jobs and put them on an event stack, we can make a plan to simulate the $G/G/1$ queue.

Replace the while loop of the previous exercise by the code below. Observe that we split events into two types[2]: arrivals and departures.

```
while stack:
    time, job, typ = heappop(stack)
    if typ == ARRIVAL:
        handle_arrival(time, job)
    else:
        handle_departure(time, job)
```

1.7. With the above idea to make a while loop, make a list of things that have to take place at an arrival event (in particular, what should happen when the server is busy at an arrival epoch, what should happen if the server is idle?) and at a departure event (in particular, what if the queue is empty, or not empty)?

Turn your ideas into code. It's not a problem if you spend some time on this; you will learn a lot in the process.

s.1.7. We need a server object to keep track of the state of the server, and we also need a list to queue the jobs.

```
class Server:
    def __init__(self):
        self.busy = False

server = Server()
```

[2] The word `type` is a reserved word in python, hence I use `typ` instead.

```
queue = []
served_jobs = [] # used for statistics

def start_service(time, job):
    server.busy = True
    job.departure_time = time + job.service_time
    heappush(stack, (job.departure_time, job, DEPARTURE))

def handle_arrival(time, job):
    job.queue_length_at_arrival = len(queue)
    if server.busy:
        queue.append(job)
    else:
        start_service(time, job)

def handle_departure(time, job):
    server.busy = False
    if queue: # queue is not empty
        next_job = queue.pop(0) # pop oldest job in queue
        start_service(time, next_job)
```

1.8. Once you have all your code (or you understand the code in the solutions) run an experiment with 10 jobs with exponentially distributed inter-arrival times with $\lambda = 2$ and service times $\sim Exp(\mu)$ with $\mu = 3$.

s.1.8. Here is an experiment.

```
def experiment_2():
    labda = 2.0
    mu = 3.0
    rho = labda / mu
    F = expon(scale=1.0 / labda)  # interarrival time distributon
    G = expon(scale=1.0 / mu)  # service time distributon
    num_jobs = 10  # too small, change it to a larger number, and rerun the experiment

    time = 0
    for i in range(num_jobs):
        time += F.rvs()
        job = Job()
        job.arrival_time = time
        job.service_time = G.rvs()
        heappush(stack, (job.arrival_time, job, ARRIVAL))

    while stack:
        time, job, typ = heappop(stack)
        if typ == ARRIVAL:
            handle_arrival(time, job)
        else:
            handle_departure(time, job)
            served_jobs.append(job)

    tot_queue = sum(j.queue_length_at_arrival for j in served_jobs)
    av_queue_length = tot_queue / len(served_jobs)
    print("Theoretical avg. queue length: ", rho * rho / (1 - rho))
    print("Simulated avg. queue length:", av_queue_length)
```

```
    tot_sojourn = sum(j.sojourn_time() for j in served_jobs)
    av_sojourn_time = tot_sojourn/len(served_jobs)
    print("Theoretical avg. sojourn time:", (1./labda)*rho/(1-rho))
    print("Simulated avg. sojourn time:", av_sojourn_time)

experiment_2()
```

1.3 *Simple tests*

As a general observation, testing code is very important. For this reason, before we can apply our simulator to real situations, we should apply it to some simple cases that we can analyze with theory. Of course the results of the simulation and the models should coincide.

1.9. Run a simulation for the $M/M/1$ queue with $\lambda = 2$ and $\mu = 3$ with 100 jobs and compute the average queue length. Then extend to 1000, and then to 10^5. Compare it to the theoretical expected queue length of the $M/M/1$ queue at arrival moments. Do the same thing for the average sojourn time.

h.1.9. You might want to check Exercise 1.6 to see how to generate the jobs.

s.1.9. Now we see that we needed to store the jobs in `served_jobs`. Set `num_jobs=100` in the code of Exercise 1.6. Put the following code at the end of the simulator to compute the statistics.

```
def experiment_2a():
    labda = 2.0
    mu = 3.0
    rho = labda / mu
    F = expon(scale=1.0 / labda)  # interarrival time distributon
    G = expon(scale=1.0 / mu)  # service time distributon
    num_jobs = 1000

    time = 0
    for i in range(num_jobs):
        time += F.rvs()
        job = Job()
        job.arrival_time = time
        job.service_time = G.rvs()
        heappush(stack, (job.arrival_time, job, ARRIVAL))

    while stack:
        time, job, typ = heappop(stack)
        if typ == ARRIVAL:
            handle_arrival(time, job)
        else:
            handle_departure(time, job)
            served_jobs.append(job)

    tot_queue = sum(j.queue_length_at_arrival for j in served_jobs)
    av_queue_length = tot_queue / len(served_jobs)
    print("Theoretical avg. queue length: ", rho * rho / (1 - rho))
    print("Simulated avg. queue length:", av_queue_length)

    tot_sojourn = sum(j.sojourn_time() for j in served_jobs)
    av_sojourn_time = tot_sojourn/len(served_jobs)
```

```
    print("Theoretical avg. sojourn time:", (1./labda)*rho/(1-rho))
    print("Simulated avg. sojourn time:", av_sojourn_time)

experiment_2a()
```

Now run the simulator.

You should see that you need a real large amount of jobs to obtain a good estimate for the (theoretical) expected queue length[3].

1.10. Thus, let us get further confidence in our $G/G/1$ simulator by specializing it to the $D/D/1$ queue. Check the documentation of the `uniform` distribution in `scipy.stats` to see what the following code does.

```
from scipy.stats import uniform

stack = [] # this is the event stack
queue = []
served_jobs = [] # used for statistics

def experiment_3():
    labda = 2.0
    mu = 3.0
    rho = labda / mu
    F = uniform(3, 0.00001)
    G = uniform(2, 0.00001)
    num_jobs = 10

    time = 0
    for i in range(num_jobs):
        time += F.rvs()
        job = Job()
        job.arrival_time = time
        job.service_time = G.rvs()
        heappush(stack, (job.arrival_time, job, ARRIVAL))

    while stack:
        time, job, typ = heappop(stack)
        if typ == ARRIVAL:
            handle_arrival(time, job)
        else:
            handle_departure(time, job)
            served_jobs.append(job)

    for j in served_jobs:
        print(j)


experiment_3()
```

Then use this in our simulation. What happens? What happens if you reverse the 2 and the 3 in F and G?

s.1.10. With `F=uniform(3, 0.00001)` the job inter-arrival times are nearly 3. Likewise, the service times are nearly 2. Hence, arriving customers will always see an empty queue, implying that the

[3] I have to admit that I don't understand why (at least in my simulation) I need about 10^5 jobs to get a good estimate.

average queue length upon arrival equals zero. When you reverse the 2 and 3, the queue should build up.

1.4 *Comparison with the $M/G/1$ queue*

As a second set of tests, we can compare the results of our simulator with those obtained for the $M/G/1$ queue. For this queue, we have a closed-form expression for the average waiting time and an algorithm to compute the queue-length distribution. So, we can, and should, use these theoretical results as further tests.

Assume that demand arrives as a Poisson process with rate $\lambda = 1/3$ per minute, and that the service distribution is $U[1,3]$ minute.

1.11. Implement the Pollaczek-Khinchine equation in python and use this to compute the expected queue length.

h.1.11. Build a function with arguments `labda` and `G`, where `G` represents a distribution. Next use `G.var()` and `G.mean()` to compute c_e^2.

s.1.11. Here is simple implementation.

```
def pollaczek_khinchine(labda, G):
    ES = G.mean()
    rho = labda*ES
    ce2 = G.var()/ES/ES
    EW = (1.+ce2)/2 * rho/(1-rho)*ES
    return EW


labda = 1./3
F = expon(scale=1./labda)  # interarrival time distributon
G = uniform(1, 2)

print("PK: ", labda*pollaczek_khinchine(labda,G))
```

1.12. Estimate the average queue length with the our $G/G/1$ simulator and compare the results with the theoretical result obtained from the PK-formala. Note that this is another test of our implementation.

h.1.12. Reset all data and clear all lists.

s.1.12. To ensure that we do not keep old data, we have to reset all lists.

```
stack = [] # this is the event stack
queue = []
served_jobs = [] # used for statistics

def test_mg1():
    job = Job()
    labda = 1.0 / 3
    F = expon(scale=1.0 / labda)  # interarrival time distributon
    G = uniform(1, 3)
    print("ES: ", G.mean(), "rho: ", labda * G.mean())

    num_jobs = 1000

    time = 0
```

```
    for i in range(num_jobs):
        time += F.rvs()
        job = Job()
        job.arrival_time = time
        job.service_time = G.rvs()
        heappush(stack, (job.arrival_time, job, ARRIVAL))

    while stack:
        time, job, typ = heappop(stack)
        if typ == ARRIVAL:
            handle_arrival(time, job)
        else:
            handle_departure(time, job)
            served_jobs.append(job)

    tot_queue = sum(j.queue_length_at_arrival for j in served_jobs)
    av_queue_length = tot_queue / len(served_jobs)
    print("Theoretical avg. queue length: ", labda * pollaczek_khinchine(labda, G))
    print("Simulated avg. queue length:", av_queue_length)

    tot_sojourn = sum(j.sojourn_time() for j in served_jobs)
    av_sojourn_time = tot_sojourn / len(served_jobs)
    print("Theoretical avg. sojourn time: ", pollaczek_khinchine(labda, G) + G.mean())
    print("Avg. sojourn time:", av_sojourn_time)

test_mg1()
```

We used Little's law to relate the average queue length and the average waiting time.

1.5 *The check-in process at an airport*

Now that we have a simulator—and we tested it, although not sufficiently well to use it for real applications—we can apply it to analyze the queueing behavior at a check-in process at an airport. For ease we focus here on a single server desk, in the next tutorials we extend the simulator to more realistic scenarios.

We need to model the arrival process and the service times. With regard to the arrivals, we know that a plane has a certain capacity. Suppose that we deal with a small plane with just 60 seats, and that the 60 customers arrive uniformly distributed over 2 hours. The service time distribution is $\sim U[1,3]$ minute.

1.13. How can we simulate such an arrival process?

s.1.13. We simulate 60 uniformly distributed arrival times $\sim U[0,120]$. Then we sort them so that the customers arrive in order and store them in an array.

```
import numpy as np

F = np.sort(uniform(0, 120).rvs(60))
```

1.14. Estimate the average queue length with our $G/G/1$ simulator. Use the `Counter` class to count the queue length distribution, and comment on the result.

h.1.14. Reset all data and clear all lists.

s.1.14. We need to start with importing the `Counter` class.

```
from collections import Counter

stack = [] # this is the event stack
queue = []
served_jobs = [] # used for statistics

def check_in():
    num_jobs = 60
    F = np.sort(uniform(0, 120).rvs(num_jobs))
    G = uniform(1, 3)

    for i in range(num_jobs):
        job = Job()
        job.arrival_time = F[i]
        job.service_time = G.rvs()
        heappush(stack, (job.arrival_time, job, ARRIVAL))

    while stack:
        time, job, typ = heappop(stack)
        if typ == ARRIVAL:
            handle_arrival(time, job)
        else:
            handle_departure(time, job)
            served_jobs.append(job)

    tot_queue = sum(j.queue_length_at_arrival for j in served_jobs)
    av_queue_length = tot_queue / len(served_jobs)
    print("Simulated avg. queue length:", av_queue_length)

    tot_sojourn = sum(j.sojourn_time() for j in served_jobs)
    av_sojourn_time = tot_sojourn / len(served_jobs)
    print("Avg. sojourn time:", av_sojourn_time)

    c = Counter([j.queue_length_at_arrival for j in served_jobs])
    print("Queue length distributon, sloppy output")
    print(sorted(c.items()))


check_in()
```

1.15. In my simulation output 2 customers see a queue length of 15. That is quite large. Lets plot the queue length at arrival moments to see whether that explains our results. This is the code you can use for this.

```
import matplotlib.pyplot as plt

x = [job.arrival_time for job in served_jobs]
y = [job.queue_length_at_arrival for job in served_jobs]

plt.plot(x, y, "o")
plt.show()
```

What do you see?

s.1.15. In this specific sample path, most jobs appear to arrive at the end of the 2 hours. And indeed, when all people arrive late, we'll see a big queue.

1.16. Change the service time distribution to $\sim U[1,2]$ to see whether that reduces the queue lengths. Suppose also that we open the desk a bit longer, half an hour say. What is the result of these changes?

s.1.16. Here is the code.

```
stack = [] # this is the event stack
queue = []
served_jobs = [] # used for statistics

def check_in_2():
    num_jobs = 60
    F = np.sort(uniform(0, 150).rvs(num_jobs))
    G = uniform(1, 2)

    for i in range(num_jobs):
        job = Job()
        job.arrival_time = F[i]
        job.service_time = G.rvs()
        heappush(stack, (job.arrival_time, job, ARRIVAL))

    while stack:
        time, job, typ = heappop(stack)
        if typ == ARRIVAL:
            handle_arrival(time, job)
        else:
            handle_departure(time, job)
            served_jobs.append(job)

    tot_queue = sum(j.queue_length_at_arrival for j in served_jobs)
    av_queue_length = tot_queue / len(served_jobs)
    print("Simulated avg. queue length:", av_queue_length)

    tot_sojourn = sum(j.sojourn_time() for j in served_jobs)
    av_sojourn_time = tot_sojourn / len(served_jobs)
    print("Avg. sojourn time:", av_sojourn_time)

    x = [job.arrival_time for job in served_jobs]
    y = [job.queue_length_at_arrival for job in served_jobs]

    plt.plot(x, y, "o")
    plt.show()
check_in_2()
```

This is much better.

1.6 *Extensions*

Here are some final, more general, questions to deepen your understanding of queues systems and simulation.

1.17. Up to now we studied the $G/G/1$ FIFO (First In First Out) queue. How to change the code to simulate a LIFO (Last In First Out) queue?

s.1.17. This is really easy: change the line `queue.pop(0)` by `queue.pop()`.

1.18. In a priority queue jobs belong to a priority class. Jobs with higher priority are served before jobs with lower priority, and within one priority class jobs are served in FIFO sequence. A concrete example of a priority queue is the check-in process of business and economy class customers at airports. How would you build a $G/G/1$ priority queue?

h.1.18. We have used a good data structure already. How should this be applied?

s.1.18. As in the LIFO example, we only have to change the data structure. First, we give each job an extra attribute corresponding to its priority. Then we use a heap to store the jobs in queue. Specifically, change the line with `queue.append(job)` by `heappush(queue, (job.priority, job))` and `queue.pop(0)` by `heappop(queue)`.

1.19. We implemented the queue as a python list. Why is a deque[4] a more efficient data structure to simulate a queue? Why is it important to know the efficiency of the data structures and algorithms you use?

h.1.19. Search the web on `python` and `deque`.

s.1.19. In a python list the `pop(0)` function is an $O(n)$ operation, where n is the number of elements in the list. In a deque appending and removing items to either end of the deque is an $O(1)$ operation.

When you do large scale simulations, involving many hours of simulation time, all inefficiencies build up and can make the running time orders of magnitude longer. A famous example is the sorting of numbers. A simple, but stupid, sorting algorithm is $O(n^2)$ while a good algorithm is $O(n \log n)$. The sorting time of 10^6 numbers is dramatically different.

For our code, add the line

```
from collections import deque
```

and replace `queue = []` by

```
queue = deque()
```

and `queue.pop(0)` by

```
queue.popleft()
```

1.20. Recall that in Exercise 1.6 we built all jobs before the queueing simulation starts. Why is this not a good decision? Note that in the simulation of $G/G/c$ queue below we will generate jobs only when needed.

s.1.20. Typically, generating all jobs at the start is not a real good idea. When running large simulations the amount of computer memory required to store all this data grows out of hand. Also our `served_jobs` list is not memory efficient.

1.7 *Summary*

1.21. What have you learned in this tutorial?

s.1.21. Topics learned.

1. Efficient data structures such as heap queues. In general, it is important to have some basic knowledge of algorithmic complexity.
2. Event stacks to organize the tracking of events in time. With the concept of event stack, you now know how discrete event simulators work.
3. The simulation of the $G/G/1$ queue. We can use simulation to analyze a practical case and come up with concrete recommendations on how to improve the system.

[4] Search the web to understand what this is.

1.22. Can you make the simulator more realistic?

s.1.22. It would be interesting to change the demand process such that the arrival rate becomes time-dependent. That would make the case more realistic, and would make the simulator more useful. In fact, the theoretical models nearly always assume that the arrival and service time distribution are constant. When these become dynamic, simulation is the way to go.